UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

MIGRATION OF NETWORKS

Bruno Miguel Costa Nunes

MESTRADO EM ENGENHARIA INFORMÁTICA

Dissertação orientada por:
Prof. Doutor Bernardo Luís da Silva Ferreira
e co-orientada pelo Prof. Doutor Fernando Manuel Valente Ramos

2021

Agradecimentos

Gostava de agradecer a toda a gente que me acompanhou durante este muito longo processo, que nunca me deixou desistir e sempre me relembrou das minhas responsabilidades.

Queria deixar um especial agradecimento à minha família pela paciência que teve para comigo, aos meus colegas de curso que me acompanharam até aqui, aos colegas do Lasige que tanto me ensinaram e, finalmente, ao meu orientador, Professor Doutor Fernando Ramos, por ter aguentado comigo ao longo de todo este tempo.

Ao meu falecido avô.

Resumo

A forma como os recursos computacionais são geridos, mais propriamente os alojados nos grandes centros de dados, tem vindo, nos últimos anos, a evoluir. As soluções iniciais que passavam por aplicações a correr em grandes servidores físicos, comportavam elevados custos não só de aquisição, mas também, e principalmente, de manutenção. A razão chave por trás deste facto prendia-se em grande parte com uma utilização largamente ineficiente dos recursos computacionais disponíveis. No entanto, o surgimento de tecnologias de virtualização de servidores foi o volte-face necessário para alterar radicalmente o paradigma até aqui existente. Isto não só levou a que os operadores dos grandes centros de dados pudessem passar a alugar os seus recursos computacionais, criando assim uma interessante oportunidade de negócio, mas também permitiu potenciar (e facilitar) negócios dos clientes. Do ponto de vista destes, os benefícios são evidentes: poder alugar recursos, num modelo pay-as-you-go, evita os elevados custos de capital necessários para iniciar um novo serviço. A este novo conceito baseado no aluguer e partilha de recursos computacionais a terceiros dá-se o nome de computação em nuvem ("cloud computing").

Como referimos anteriormente, nada disto teria sido possível sem o aparecimento de tecnologias de virtualização, que permitem o desacoplamento dos serviços dos utilizadores do hardware que os suporta. Esta tecnologia tem-se revelado uma ferramenta fundamental na administração e manutenção dos recursos disponíveis em qualquer centro de dados. Por exemplo, a migração de máquinas virtuais facilita tarefas como a manutenção das infraestruturas, a distribuição de carga, a tolerância a faltas, entre outras primitivas operacionais, graças ao desacoplamento entre as máquinas virtuais e as máquinas físicas, e à consequente grande mobilidade que lhes é assim conferida.

Atualmente, muitas aplicações e serviços alojados na nuvem apresentam dimensão e complexidade considerável. O serviço típico é composto por diversos componentes que se complementam de forma a cumprir um determinado propósito. Por exemplo, diversos serviços são baseados numa topologia de vários níveis, composta por múltiplos servidores web, balanceadores de carga e bases de dados distribuídas e replicadas. Daqui resulta uma forte ligação e dependência dos vários elementos deste sistema e das infraestruturas de comunicação e de rede que os suportam.

Esta forte *dependência da rede* vem limitar grandemente a flexibilidade e mobilidade das máquinas virtuais, o que, por sua vez, restringe inevitavelmente o seu reconhe-

cido potencial. Esta dependência é particularmente afetada pela reduzida flexibilidade que a gestão e o controlo das redes apresentam atualmente, levando a que o processo de migração de máquinas virtuais se torne num demorado processo que apresenta restrições que obrigam à reconfiguração da rede, operação esta que, muitas vezes, é assegurada por um operador humano (de que pode resultar, por exemplo, a introdução de falhas).

Num cenário ideal, a infraestrutura de redes de que depende a comunicação entre as máquinas virtuais seria também ela *virtual*, abstraindo os recursos necessários à comunicação, o que conferiria à globalidade do sistema uma maior flexibilidade e mobilidade que, por sua vez, permitiria a realização de uma migração conjunta das referidas máquinas virtuais e da infraestrutura de rede que as suporta.

Neste contexto, surgem as redes definidas por software (SDN) [34], uma nova abordagem às redes de computadores que propõe separar a infraestrutura responsável pelo encaminhamento do tráfego (o plano de dados) do plano de controlo, planos que, até aqui, se encontravam acoplados nos elementos de rede (switches e routers). O controlo passa assim para um grupo de servidores, o que permite criar uma centralização lógica do controlo da rede. Uma SDN consegue então oferecer uma visão global da rede e do seu respetivo estado, característica fundamental para permitir o desacoplamento necessário entre a infraestrutura física e virtual.

Recentemente, várias soluções de virtualização de rede foram propostas (e.g., VMware NSX [5], Microsoft AccelNet [21] e Google Andromeda [2]), ancoradas na centralização oferecida por uma SDN. No entanto, embora estas plataformas permitam virtualizar a rede, nenhuma delas trata o problema da migração dos seus elementos, limitando a sua flexibilidade.

O objetivo desta dissertação passa então por implementar e avaliar soluções de migração de redes recorrendo a SDNs. A ideia é migrar um dispositivo de rede (neste caso, um switch virtual), escolhido pelo utilizador, de modo transparente, quer para os serviços que utilizam a rede, evitando causar disrupção, quer para as aplicações de controlo SDN da rede. O desafio passa por migrar o estado mantido no switch de forma consistente e sem afetar o normal funcionamento da rede.

Com esse intuito, implementámos e avaliámos três diferentes abordagens à migração (*freeze and copy*, *move* e *clone*) e discutimos as vantagens e desvantagens de cada uma. É de realçar que a solução baseada em clonagem se encontra incorporada como um módulo do virtualizador de rede Sirius [12].

Palavras-chave: Cloud, Virtualização, Migração, Redes definidas por software

Abstract

The way computational resources are managed, specifically those in big data centers, has been evolving in the last few years. One of the big stepping-stones for this was the emergence of server virtualization technologies that, given their ability to decouple software from the hardware, allowed for big data center operators to rent their resources, which, in its turn, represented an interesting business opportunity for both the operators and their potential customers. This new concept that consists in renting computational resources is called cloud computing. Furthermore, with the possibility that later arose of live migrating virtual machines, be it by customer request (for example, to move their service closer to the target consumer) or by provider decision (for example, to execute scheduled rack maintenances without downtimes), this new paradigm presented really strong arguments in comparison with traditional hosting solutions.

Today, most cloud applications have considerable dimension and complexity. This complexity results in a strong dependency between the system elements and the communication infrastructure that lays underneath. This strong *network dependency* greatly limits the flexibility and mobility of the virtual machines (VMs). This dependency is mainly due to the reduced flexibility of current network management and control, turning the VM migration process into a long and error prone procedure.

From a network's perspective however, software-defined networks (SDNs) [34] manage to provide tools and mechanisms that can go a long way to mitigate this limitation. SDN proposes the separation of the forwarding infrastructure from the control plane as a way to tackle the flexibility problem. Recently, several network virtualization solutions were proposed (e.g., VMware NSX [5], Microsoft AccelNet [21] and Google Andromeda [2]), all supported on the logical centralization offered by an SDN. However, while allowing for network virtualization, none of these platforms addressed the problem of migrating the virtual networks, which limits their functionality.

The goal of this dissertation is to implement and evaluate network migration solutions using SDNs. These solutions should allow for the migration of a network element (a virtual switch), chosen by the user, transparently, both for the services that are actively using the network and for the SDN applications that control the network. The challenge is to migrate the virtual element's state in a consistent manner, whilst not affecting the normal operation of the network. With that in mind, we implemented and evaluated three

different migration approaches (*freeze and copy*, *move* and *clone*), and discussed their respective advantages and disadvantages. It is relevant to mention that the cloning approach we implemented and evaluated is incorporated as a module of the network virtualization platform Sirius [12].

Keywords: Cloud, Virtualization, Migration, Software-defined network

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The way computational resources are managed, specifically those in big data centers, has been evolving in the last few years. Initially, the solutions consisted in applications running in big physical servers, however, this brought considerable costs. The key piece that changed the state-of-affairs was the emergence of server virtualization technologies. These allowed big data center operators to rent their resources, which represented an interesting business opportunity for both the operators and their potential customers. From a customer perspective, the benefits are also evident: being able to rent resources in a pay-as-you-go model represents a considerably cheaper alternative to buying the necessary hardware up-front, to support a new service with yet unpredictable usage patterns. This new concept that consists in renting computational resources is called, nowadays, cloud computing.

Today, a great amount of the existing applications that run in cloud environments presents considerable dimension and complexity. The typical service is composed of several components that complement each other in order to fulfill a common goal, e.g., numerous services are based on multi-level topologies, with a layer of web servers, another of load balancers, and distributed and replicated databases. This complexity results in a strong dependency between the system elements and the communication infrastructure that lays underneath. This strong *network dependency* greatly limits the flexibility and mobility of the virtual machines (VMs). This dependency is particularly affected by the reduced flexibility that the current network management and control tools and techniques provide, turning the VM migration process in a long and error prone procedure.

1.1 Motivation

The tight dependence between modern applications and systems and their underlying network makes the latter the limiting element in improving the efficiency and flexibility of cloud infrastructures. Ideally, the entire network infrastructure of which VMs are dependent should be itself virtual, abstracting the necessary communication resources, which

would result in a globally more flexible and adaptable infrastructure.

Software-defined networks [34] have recently been demonstrated to be the enabler for this form of network virtualization. SDN advocates splitting the infrastructure responsible for traffic forwarding (data plane) from the control plane (in traditional architectures these components are strongly coupled). The control now typically runs in one (or more) servers, the SDN controllers, resulting in logically centralized network control. Therefore, the controller has a global vision of the network and of its state, which is the key in enabling the decoupling of the virtual network from the substrate infrastructure.

Recently, we've witnessed the emergence of several network virtualization platforms (e.g., VMware NSX [5], Microsoft AccelNet [21] and Google Andromeda [2]) that make use of SDNs to achieve this advanced form of network virtualization.

While these platforms can serve as a framework to enable network migration, none has actually provided a working solution to this problem. Therefore, there remains a need to study technical and algorithmic solutions to tackle the network migration challenge.

1.2 Goals and Challenges

The goal of this thesis is to investigate algorithms to perform the migration of a network component (namely, virtual switches), *transparently*. By transparent we mean that the communicating VMs should not be able to distinguish the network behaviour during migration from normal operation. This, however, does not mean there cannot exist any kind of disturbance during the migration, but rather that those disturbances also occur under normal operation. For example, slight variance of latency or throughput and events such as packet loss or out of order delivery do not break transparency as they can occur in a best effort network.

Our goal is to explore the advantages and disadvantages of different algorithms to understand the trade-offs that the transparency requirement induces.

We will focus mainly on the following problems:

- Migrating a network switch should ideally have no impact in user applications and services. In particular, network performance, as experienced by the applications that communicate across the network, should not significantly degrade (in terms of throughput, latency, and packet loss);

- In an SDN context several control applications run on the controller to support network operation. The migration of a switch should therefore be transparent to these control applications (e.g., packet counters, flow information, and related state should be maintained consistently).

1.3 Contributions

This work's contributions are the following:

- The design and implementation of three techniques for migration of a switch, as applications that run on top of the Floodlight SDN controller [1]. The choice of techniques was made to explore the main trade-offs of the design space: one algorithm favors transparency (*clone*); another favors migration speed (*freeze and copy*); and another attempts to be a middle ground between the two (*move*).
- Evaluation of the different migration variants under analysis, and discussion of the trade-offs.

In addition, we have also implemented three additional Floodlight modules that provide support to the migration problem (but can also be used stand-alone), as well as some bug fixing on existing Floodlight modules (reported to the community). Finally, the *clone* method developed was incorporated into the Sirius multi-cloud network hypervisor [12].

1.4 Document Structure

The remainder of this document is structured as follows:

- Chapter 2 – presents background on the areas deemed relevant to this work's topic, as well as existing related work;
- Chapter 3 – is an in depth look into this work's main contribution, namely the design and implementation of the migration mechanisms under study;
- Chapter 4 – presents the evaluation of the network migration algorithms implemented, and discusses the trade-offs involved;
- Chapter 5 – concludes this document with a summary of the dissertation, including some ideas for future work.

Chapter 2

Related Work

This chapter presents the background to this dissertation and closely related work.

We start, in Section 2.1, by explaining server virtualization. Afterwards, in Section 2.2, we present techniques for live migration of virtual machines, explaining what is preventing this technique from going further as a universally used resource management tool. Next, in Section 2.3, we turn our focus to the network side, introducing Software-defined networks (SDN), and how this new paradigm enabled advanced forms of network virtualization, some of which are presented afterwards, in Section 2.4. In Section 2.5, we detail related work on network migration, and we close with some final remarks in Section 2.6.

2.1 Server Virtualization

Virtualization consists in creating an abstraction layer between the hardware and the software running on top of it. As a result, the software becomes less dependent of the underlying infrastructure where it is executed, with clear gains in flexibility for the infrastructure, easing the management and maintenance of its resources.

The benefits, namely to data center environments, include resource sharing (a single machine providing resources for several virtual machines, or VMs), simplifying an application's deployment (since it will be running on top of a VM, it does not need to be adapted to the specific hardware where it will run), reduction of deployment failures (by enabling testing on the exact environment encountered in production), avoidance of vendor lock-in (the software can run any hardware capable of running the virtual machine manager (VMM)), while guaranteeing resource isolation (the VMM ensures that each VM has the resources it needs available, without conflicting with any other VM).

There are different virtualization approaches (Figure 2.1). Full virtualization [13] introduces a new abstraction layer, the hypervisor, over the hardware, allowing VMs to run different guest operating systems on top of a single server. By contrast, kernel-based approaches [31] modify an existing OS at the kernel level, in order to include the necessary mechanisms to enable multiple VMs to run in parallel as a lightweight virtualization

mechanism. However, they are restricted to the same operating system (OS).

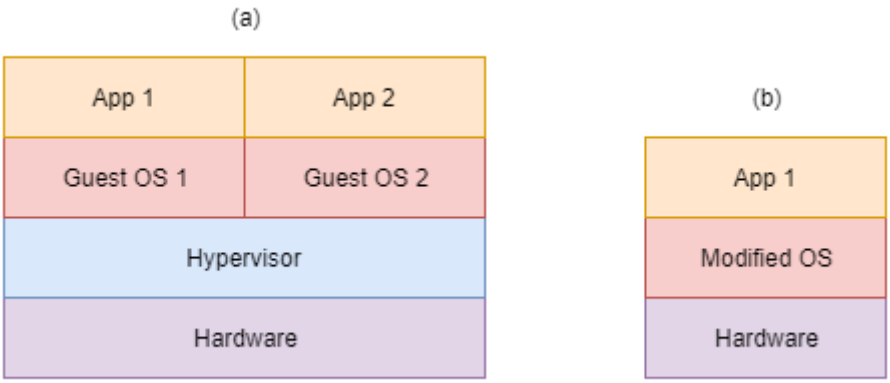

Figure 2.1: Hypervisor-based virtualization (a). Kernel-based virtualization (b).

A third type is nested virtualization (Figure 2.2). It essentially consists in the addition of another abstraction layer placed over the already existing hypervisor.

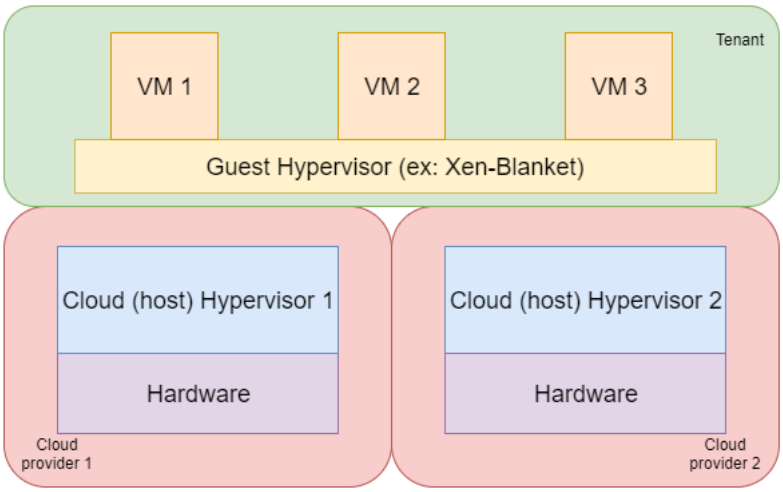

Figure 2.2: Example of nested virtualization.

This sort of virtualization can be especially useful to run applications in third party clouds, since it concedes a hypervisor-level control to the cloud user (that now also becomes a tenant), and allows for implementation of a variety of new features without requiring provider support. For example, it enables VM migration (to be discussed in further detail in the next section) amongst clouds of different providers. An example of this kind of solution is Xen-Blanket [45]. This solution provides the user with hypervisor-level control and a set of features that is independent from the cloud providers that form the base infrastructure. The main enabler is the blanket layer (Figure 2.3), which contains blanket drivers that, essentially, are responsible for interfacing with the different clouds.

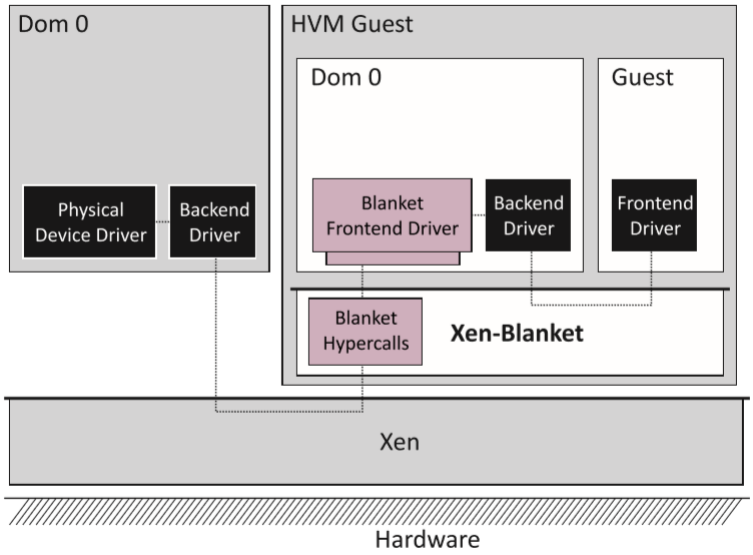

Figure 2.3: Structure of Xen-Blanket, from [45].

2.2 Live Migration of Virtual Machines

Data center and cloud administration is a challenging task, requiring careful and slow planning and involving human intervention. The introduction of virtualization significantly improved the situation, but it was not enough. For example, even though virtual machines greatly improved resource usage in data centers, virtualization alone was still far from optimal, from a global perspective. The emergence of techniques for live migration of virtual machines was therefore the key to take infrastructure operation and management to a new level.

The earliest work on live migration of VMs was presented by Clark et al. [17]. The proposed solution was built on top of Xen's hypervisor, targeted to local area networks (LANs). The main challenge was minimizing the downtime. The technique employed guaranteed transparency by not requiring guest operating systems' participation in the migration process, that remains unchanged.

The proposed migration algorithm consists of six different stages. The first, "stage 0" or pre-migration, has the sole purpose of pre-selecting the destination host of the next migration, aiming to speed up the next stages' execution. This is an optional stage. Next is the reservation stage, crucial for the process. Here the algorithm validates if the destination host has the resources necessary to host the migrated VM. If so, the resources are reserved. Otherwise, the migration process is canceled. Stage 2 is the pre-copy, which consists in the iterative copy of all memory pages from the target virtual machine to the destination one. Being an iterative process, after the first copy only the pages that were modified during the previous copy are copied and sent again. This iteration will only stop when the "writable working set" is identified. This set essentially represents the memory

pages that are constantly being modified (so often that the copy process cannot keep up).

Afterwards, we get to the stop-and-copy stage, where the original VM is suspended, its network traffic is redirected to the new VM and, finally, the remaining state is transferred, resulting in a suspended copy yet consistent, in both hosts (original and new). If, for any reason, this stage fails, the original VM will return to execution. Stage 4 is the commitment, the phase when the new VM informs the original one that everything went accordingly with the plan. This results in the release of the original VM and the new one assuming the "primary" status. The last stage is called activation and basically runs some post-migration code that will reattach device drivers to this new machine and advertise the new IP addresses. The full process is depicted in Figure 2.4.

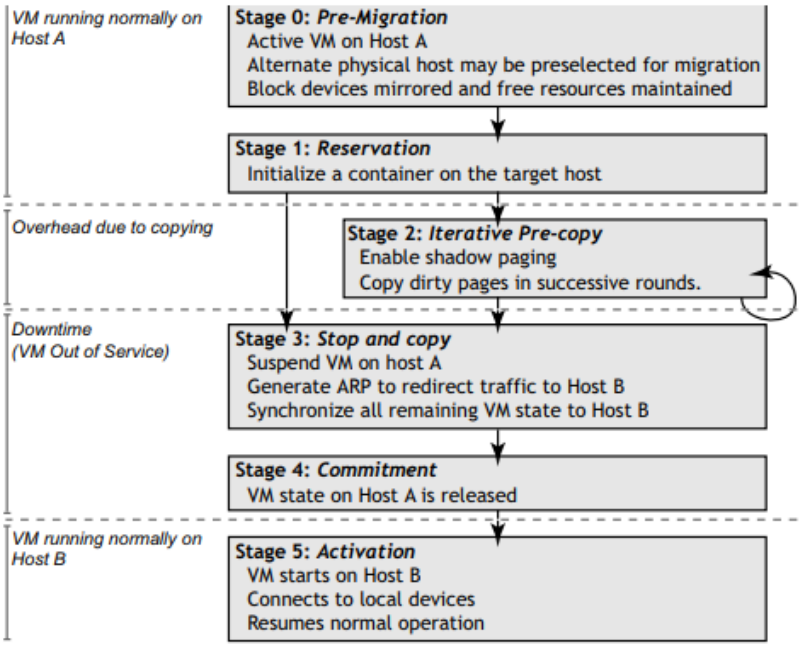

Figure 2.4: Migration timeline (from [17]).

To optimize this algorithm defining the writable working set is key. Since its identification is essentially the stopping condition for one of the fundamental stages (pre-copy), the authors analyzed several types of workload and their specific impact on the pre-copy stage in order to develop heuristics that accurately determined the optimal number of iterations. As a result, this solution minimized time and CPU usage. The authors also concluded that bandwidth usage during this stage should have enforced limits in order to control this operation's impact in the VM's normal operation.

2.2.1 Live WAN Migration of VMs

Wide area network (WAN) and LAN migrations, have some distinct features, namely the higher latency between hosts and bandwidth restrictions being, once again, key factors to

account for.

The first work to consider this problem was CloudNet, from Wood et al. [46]. The authors present a cloud computing platform that coordinates with the underlying network provider to enable seamless connectivity between enterprise and data center sites. This is justified to create a resource pool from the various clouds in order to provide flexible placement and live migration of applications across all the available sites. To achieve this, they introduced the notion of virtual cloud pools (VCP), an abstraction similar to the virtual private cloud we can find today in Amazon EC2. This abstraction allows to logically group server resources (that might be distributed across several data centers) in a single and transparent server pool.

At the network level, the solution leverages a layer-2 virtual private network (VPN) based on multi-protocol label switching (MPLS) and virtual private LAN services (VPLS). This solves the limitation of the original solution that was limited to a single IP sub-network (that arises when you consider a WAN setting), since it virtualizes the data link layer (layer 2), "placing" all resources in the same virtual LAN while, in fact, they are spread in different networks.

In order to adapt to the WAN environment, this solution proposes a slightly different migration algorithm from the original [17]. Firstly, as mentioned, it will establish a virtual connection between the VCP endpoints (origin and destination). Then, if the storage is not already shared, it transfers the disk state. Afterwards, it transfers the memory state of the original VM to the destination (in an iterative process similar to [17]) and, finally, it will stop the VM to perform the last memory and CPU state transfer. This process, shown in Figure 2.5, is therefore similar to [17], apart from the first step (virtual connections). The differences become more noticeable when we look at some of the optimizations that were made to mitigate WAN-related issues.

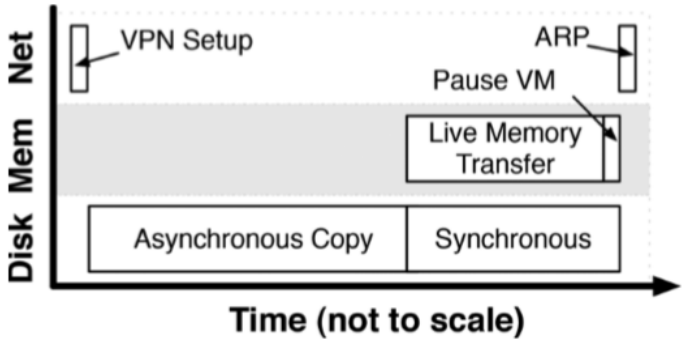

Figure 2.5: WAN migration phases (from [46]).

The first is related with the iterative step of memory transfer or, to be more precise, with its stopping. The smart stop algorithm proposed results from an analysis of the limitations of the original stopping conditions from [17], which was too costly for a WAN environment, both in terms of time and bandwidth. Therefore, they proposed a new stop-

ping condition: the iterative process would only stop if, in a given iteration, there were fewer pages to be sent than in the previous iterations. If an increasing trend was detected, the migration was aborted.

To further improve bandwidth usage, CloudNet's authors present a content-based redundancy (CBR) elimination technique. The idea is to divide the disk and memory content (each disk block or memory page) into fixed-size blocks, and hash their contents. They created a cache in both the source and destination for the block hashes. This way, before sending data, if both endpoints had the hash for the block in question, a small 32-bit index would be sent, instead of the full block.

Finally, CloudNet implements a page delta technique which is based on the fact that often only small portions of memory pages are dirtied. By taking advantage of this technique, if a page is being retransmitted, instead of sending the entire page, only the difference needs to be transmitted.

Another related work is [16], whose goal was to migrate a VM (along with its local persistent state, that is, local file system) while attempting not to interrupt any ongoing connection. In contrast to the previous work ([46]), the authors resort to dynamic DNS and tunneling in order to virtualize the network. Tunneling is used between the source VM host and the destination VM host in order to relay existing connections, to keep them alive. To ensure the new connections go to the right host, the solution updates the DNS entry immediately before the migration is concluded. This way, these connections are already established with the new host. The migration process follows the same general lines as [17, 46].

Some optimizations include using VM image templates to save bandwidth when transferring the original VM across the WAN. Similar to the CBR elimination technique or the page deltas in CloudNet, the idea is to check the differences from what is already in the template image and what is different, to transmit only the differences. Another optimization is the enforcing of a write-throttling mechanism, that is, a mechanism that slows down the write operations in the original VM (according to some thresholds) in order to reduce the differences remaining to be migrated when the VM is fully stopped.

2.2.2 Migration for Fault-tolerance

Data centers have strict requirements in terms of fault-tolerance as service unavailability or downtime can be extremely costly. For this reason, data centers operators employ several kinds of infrastructural redundancy (power, network, and computational). Nevertheless, even with these solutions in place, there may be circumstances (some predictable, others not) that may still take out part or even the totality of a data center's operation (e.g. natural disasters).

From an application point of view, fault-tolerance typically implies sophisticated re-engineering to accommodate complicated recovery logic, making fault-tolerant apps highly

complex.

The authors of Remus [18] approach this problem by proposing a mechanism that can provide high availability to existing applications, as a service. This represents a considerable advantage to existing applications since, this way, they do not require code changes to include or adapt to any complex fault-tolerant logic.

Succinctly, Remus works as depicted in Figure 2.6. At its core, it replicates the origin VM's internal state to another host, several times a second, so that, if or when the original VM fails, the backup can simply take its place (just like in a warm replica setup).

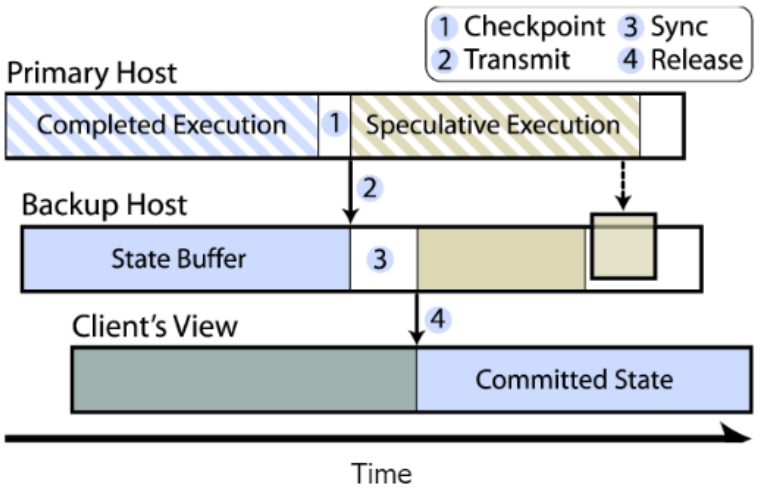

Figure 2.6: Replication process in Remus, from [18].

The way Remus performs replication is also one of its defining characteristics, and that process is visible in Figure 2.6. Instead of constantly sending the state or completely stopping the VM and sending a full state copy, which would severely affect performance, the authors implement a speculative execution mechanism and, afterward, perform an asynchronous state transfer to the backup. A key factor in the process is the fact that relevant data will only be returned to the user once the backup has confirmed it has received it for consistency. This is made possible because Remus implements buffers that retain any communication produced since the last consistent checkpoint. This way, the original VM will not stop its execution while waiting for confirmation of state replication but it also will not affect its consistency.

A different approach is taken by Bodík et al. [15]. The authors consider the inherent tradeoff between achieving high fault-tolerance and reducing bandwidth usage, since the first typically implies spreading the machines across the data center, while the second favors keeping them together (Figure 2.7).

To assist their algorithm design, the authors have thoroughly analyzed an application's communication patterns (they targeted Microsoft Bing). One important conclusion was

that the core of the network was the most congested network segment, with the remaining traffic being spread thin across the remaining of the network.

Given their findings, they designed an optimization framework that focused on bandwidth, fault-tolerance, and the number of reallocations needed from the original setup to the optimal one. From this starting point, they proposed two algorithms and proceeded with several experiments to select the best. In the end, the authors concluded that the best algorithm would reallocate the machines with a minimum of 20% bandwidth usage reduction in the network core, while improving the average worst-case survival by at least 40%.

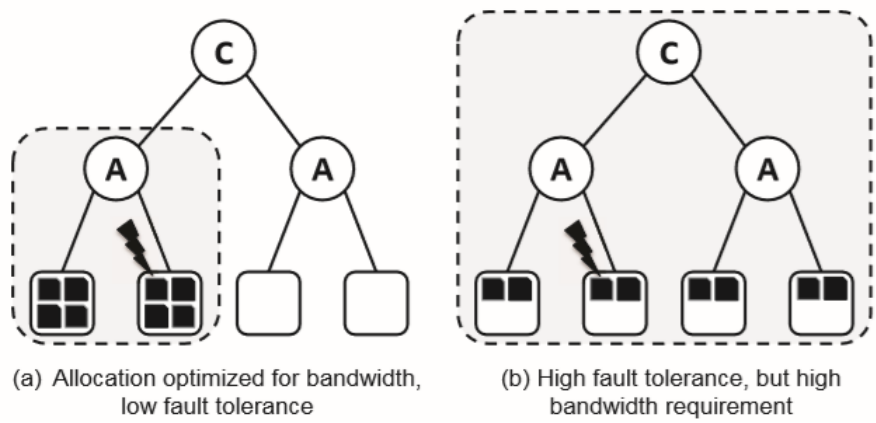

Figure 2.7: Bandwidth (a) and fault tolerance (b) optimization examples, from [15].

2.2.3 VM Placement

Live migration has, as we have shown previously, numerous benefits. However, if not properly planned and executed, it can also cause problems, including physical machine overload or network congestion. Therefore, it is important to take these aspects into consideration whenever moving any VM from one host to another.

To address these constraints, the authors of [8] proposed an algorithm that aims to consolidate heterogeneous VMs given their communication patterns. To achieve this the authors have made a few assumptions. Firstly, inter-tenant VM communication should remain small or, if possible, be inexistent; secondly, during off-peak (lower usage) periods it is acceptable to consolidate all of a tenant's VMs in a single machine; lastly, there cannot be traffic leaving the data center to reach popular resources (e.g. Google).

The algorithm starts by determining which hosts are loaded over a pre-established threshold (these are designated as undesirable destinations for migration). Then, it makes the same but for hosts under another pre-established threshold (potential destinations). Afterward, the potential destination hosts are sorted by actual load, and a set of VMs suitable for the available capacity is identified. To conclude, leveraging the VMs' communication graphs, the algorithm picks the largest connected set of VMs possible to migrate

into a single machine. This will be done until there are no longer physical machines capable of taking full sets. At this point, the larger sets are split with the help of two auxiliary partitioning algorithms and the same logic is run again, this time around, considering the newly split sets as standalone sets. This design was shown to minimize traffic between VMs hosted in different physical machines.

The migration of virtual clusters (VCs) is another situation that relies on proper arrangement. While some works focus mainly on reducing the impact caused by the migration process (e.g. [20]), [49] compares a variety of existing live migration strategies for VCs and then proposes a framework to manage these migrations.

The migration strategies proposed are concurrent migration, when a given amount of machines are simultaneously migrated; mutual migration, which consists of two clusters concurrently migrating to each other's machines; homogeneous multi-VC migration, essentially migrating multiple same-size clusters; and heterogeneous multi-VC migration, which consists of migrating multiple clusters of variable size.

The authors compared the performance of the various strategies, considering various metrics (migration time, bandwidth usage, amongst others) and concluded that concurrent migration of a big number of machines will hurt network's performance and be a slow process if faced with limited bandwidth. Furthermore, sequential migration (which this work bundles in with concurrent migration) is considered to be a better option when compared with mutual migration. In fact, mutual migration should always be avoided since it consistently results in long migration times.

2.3 Software-Defined Networking

IP networks are traditionally very rigid and hard to change. Changing a network policy usually requires a human operator to deploy or update each node's policies (Figure 2.8) by hand or using low-level scripts, in an error-prone process. Additionally, the layers of functionality of a switch or router are tightly coupled (that is, the same device concentrates both the control and data planes [34]). To address this, software-defined networks (SDN) have emerged.

2.3.1 Architecture

Software-defined networks are a new paradigm for networks that, even though they are still not globally used, promise to provide an answer to most of the traditional networks' downsides. Google and other large operators have already deployed SDNs inside their data centers [41], between data centers [26, 35], and to interconnect them with the Internet [48].

SDN presents three defining characteristics. First, the control and data planes are decoupled (which can be an enabler for innovation in the infrastructure), with the data

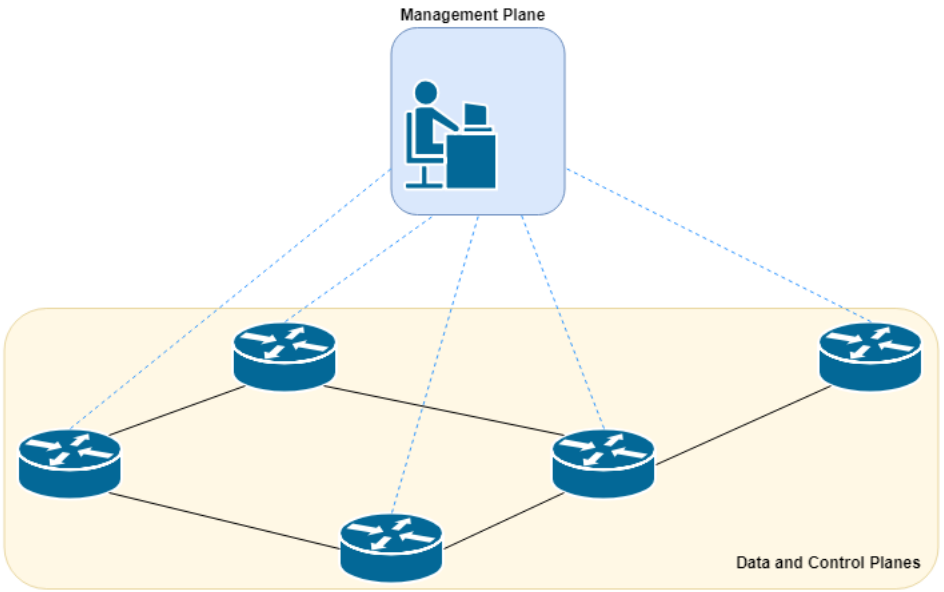

Figure 2.8: Traditional network management.

plane remaining in the switches and the control plane being transferred to a logically-centralized controller (Figure 2.9). Controllers run as a cluster of servers, and this logical centralization of the network control effectively makes it easier to observe the network's state and make decisions based on it. Second, forwarding is flow-based, which results in greater flexibility. Third, SDNs bring the ability to program the network. That is, instead of configuring a network, as in traditional networks, operators implement applications in the controller that are responsible for proactively (before any traffic reaches the network) or reactively (as the traffic reaches a device) install the necessary configurations and define the network's behavior.

As we show in Figure 2.9, to materialise the coupling of the control and data planes it is necessary a communication protocol, which is typically Openflow [38].

2.3.2 OpenFlow

Openflow [38] is the most widely adopted SDN south band protocol, used for controllers to communicate with switches. Its main purpose (and advantage) is hiding the likely existing heterogeneity of network devices, providing a common interface and, consequently, normalizing the means used to alter their functionalities.

The common workflow is as follows. When a packet arrives, it either matches some rule in the switch's forwarding table, and it executes the corresponding action or, if it did not match any rule, typically it sends an Openflow message to the controller to define the procedure. Openflow's forwarding is flow-based. This flow abstraction confers higher flexibility when it comes to handling packets.

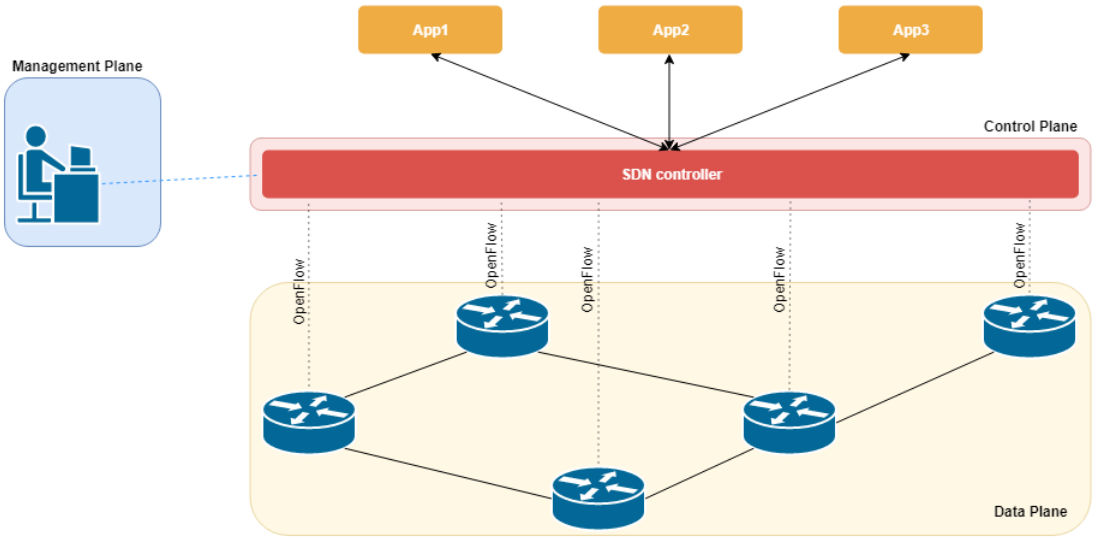

Figure 2.9: Software-defined network structure.

2.4 Network Virtualization

Traditional network virtualization solutions, such has VLANs, are limited, as they do not fully virtualize the topology and addressing of a virtual network. This limits its usefulness, for example, in data center environments. A scenario where this gets clear is when a data center workload is dependent on a given topology, which is often the case. While some services require only a L2 connectivity service, others, such as web services, require multiple tiers (load balancers, databases, etc.) and L3 routing functionality.

Ideally, the virtual network abstraction should have equivalent capabilities as the compute virtualization, that is, the virtual network should be fully decoupled from the substrate infrastructure.

In this section, we present works that go towards that goal.

2.4.1 Single-cloud Network Virtualization

VMware's Network Virtualization Platform (NVP) [32] was the first production-quality network virtualization solution. Its SDN-based architecture is built around a network hypervisor that's responsible for the network abstraction that is presented to tenants. This abstraction includes a control abstraction, giving the tenants the possibility to create and manage network elements (logical datapaths) in a fashion that emulates a physical network and a packet abstraction, which consists of any packet sent in the virtual network having the same treatment they would if this virtual network was the tenant's physical network.

The aforementioned logical datapaths are implemented on a software switch (running in every host), taking advantage of tunnels between every pair of host-hypervisors, to get

connectivity and guaranteeing also the necessary isolation. This results in transparency in the underlying network, that will only see normal IP traffic circulating and will not require any infrastructural change.

NVP has a logically centralized controller implemented as a cluster of servers responsible for the configuration of every software switch. NVP takes advantage of traffic locality by installing the packet flow rules in the kernel. This results in faster matching. To further improve throughput, NVP resorts to stateless transport tunneling (STT) for encapsulation, to enable hardware offloading mechanisms.

The second challenge was the complexity of the computation of the network's forwarding state at the controller. To tackle it, NVP uses a domain-specific declarative language (nlog) that allows for the separation of logic specification from the state machine that implements it, simplifying computation.

Finally, for scalability, NVP divides the complex computations into multiple tasks to be executed by different servers in parallel. Availability is achieved by having hot standby replicas ready in the eventuality of a failure being detected.

OpenVirteX (OVX) [9] is another SDN-based network virtualization platform. The main difference is that its goal is to virtualize a SDN. OVX can be considered a network hypervisor (based on FlowVisor [40], a network virtualization layer) that inserts an additional abstraction layer between the physical network and the SDN controller(s) (Figure 2.10). OVX fully virtualizes the network, enabling the creation of several virtual SDNs (vSDNs) running on top of a single infrastructure, providing each tenant with a network following their requisites (topology, addressing, and control).

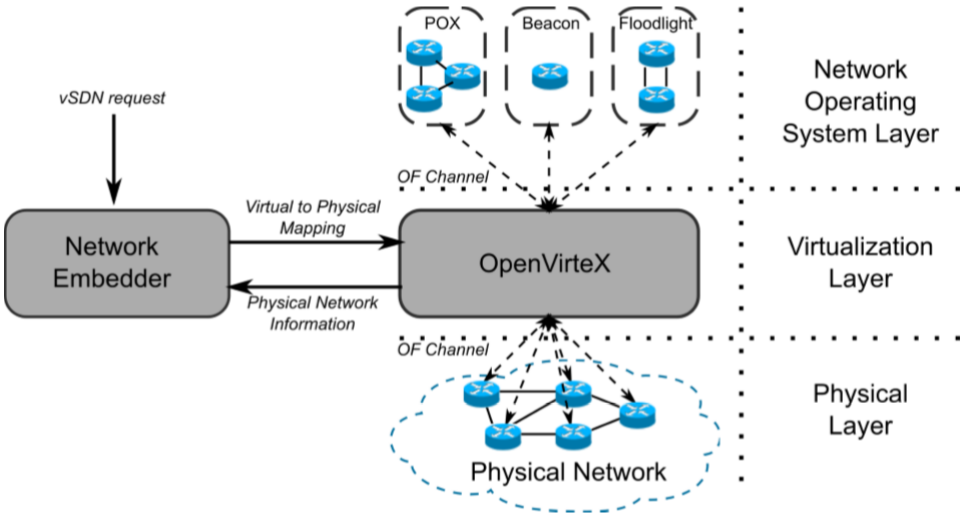

Figure 2.10: OpenVirtex architecture, from [9].

To achieve topology virtualization, OVX intercepts LLDP messages coming from the SDN controller and "forges" LLDP response packets to send back, creating an illusion of virtual links. For address virtualization, it is necessary to consider potential overlapping

IP address blocks to exist in the same physical network. Collisions are avoided with the installation of flow rules whose objective is to rewrite addresses at the edge switches of the network. Finally, to virtualize the control, OpenVirteX maps the control functions issued by the control applications for the different virtual networks onto the corresponding physical network actions, as well as rewrites them to guarantee isolation.

2.4.2 Multi-cloud Network Virtualization

While NVP [32] assumes a single cloud infrastructure and full hardware control, some works propose to extend virtualization to multiple clouds. For example, [10, 12] apply network virtualization to several clouds, either public or private, and with varying levels of infrastructural control. The motivations for this sort of solution include reduced costs, improvements in performance, reliability and security [27, 28, 29].

To deliver such system, Alaluna et al. [12] opted for a container-based virtualization approach [42], and SDN-based control. To achieve full network virtualization, the hypervisor is responsible for translating virtual events to physical ones (and vice-versa) as well as doing flow translation at the edge of the network.

XenFlow [37] is another network virtualization platform for multi-cloud scenarios. A differentiating factor is that its solution addresses the problem of quality of service (QoS) provisioning. This proposal is designed to run on commodity hardware (using Xen as base) and its architecture consists of three components: the virtual routers, a XenFlow Server and a packet forwarding module (compatible with OpenFlow).

The virtual routers run a XenFlow client whose objective is to monitor the routing and ARP tables, looking for any updates, and collecting that data. Afterwards, that data will be collected from all routers by the XenFlow server to produce a routing information base. In order to achieve virtual network isolation, XenFlow leverages VLAN tags, and for resource isolation, it associates virtual routers with Open vSwitch queues.

XenFlow also monitors the resource usage of each virtual network and calculates the best idle resources distribution between the virtual networks, so that there are less resources being wasted. This results in a more efficient resource distribuition, as well as better link usage.

VirtualWires [44] is yet another approach to network virtualization. The main innovation is a primitive proposed to allow for a user to easily connect (or disconnect) virtual network interfaces (vNICs) with a point-to-point tunnel (Figure 2.11). Its implementation not only uses Xen but also leverages Xen-Blanket [45] to be deployable across multiple clouds.

This design allows for flexibility by ensuring that, when two vNICs are connected they will remain connected even if the topology gets modified (e.g. one of the machines gets migrated to a different cloud). As referred, the vNICs are paired with layer-2-in-layer-3 tunnels, with the tunnel connections being managed by an endpoint manager residing in

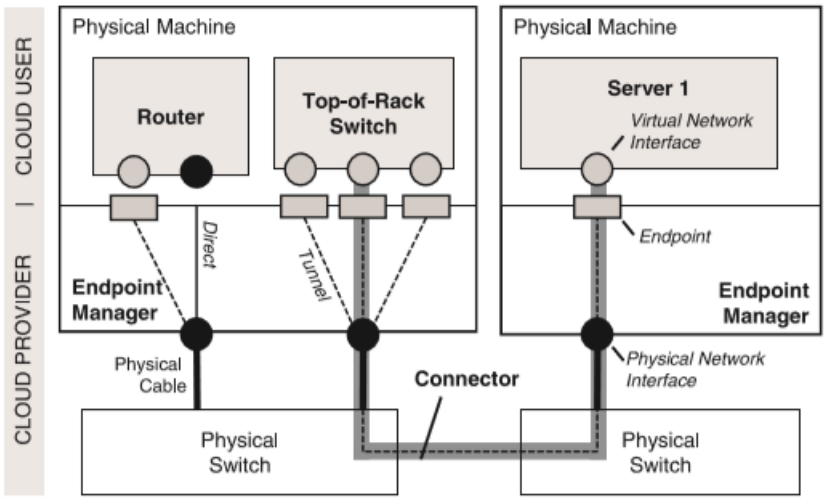

Figure 2.11: Example of a "VirtualWire", from [44].

the hypervisor. To maintain the connectors in a migration across clouds, VirtualWires introduces another component, the extender, whose purpose is to permanently maintain VPN connections with other clouds.

2.5 Network Migration

The flexibility and mobility of virtual machines is strongly limited by the network dependency of the underlying substrate resources. A relatively recent line of work has tried to address this problem by migrating the network with the VMs. We present a selection of these works in this section.

Probably the first work to address this problem is VROOM [43], a network-management primitive that allows for virtual routers to freely move between their physical counterparts while avoiding changes to the logical topology. VROOM's internals can be seen in Figure 2.12. Essentially, they split the physical router's resources into several virtual entities, creating a VM-like solution for the routers, where the functionality is separated from the hardware that runs it. However, the migration of these virtual routers can not be done in the same fashion as VM migration since the latter usually results in significant stoppage times.

Therefore, VROOM's authors suggest that data plane operations should never be interrupted, while the control plane can withstand small delays. With this in mind they decoupled the control and data planes by introducing a migration-aware interface between them (the data plane hypervisor). Besides allowing for this separation between planes, the data plane hypervisor also helps to hide the underlying data plane heterogeneity from different technologies.

With the virtual routers migratable and the control and data planes detached, the only

thing missing was actually ensuring that the target router's links were also migratable. To face this, Wang et al. decided to use programmable transport networks [7] and packet-aware transport networks [6]. These techniques, combined, allowed them the necessary link flexibility to fully migrate a router.

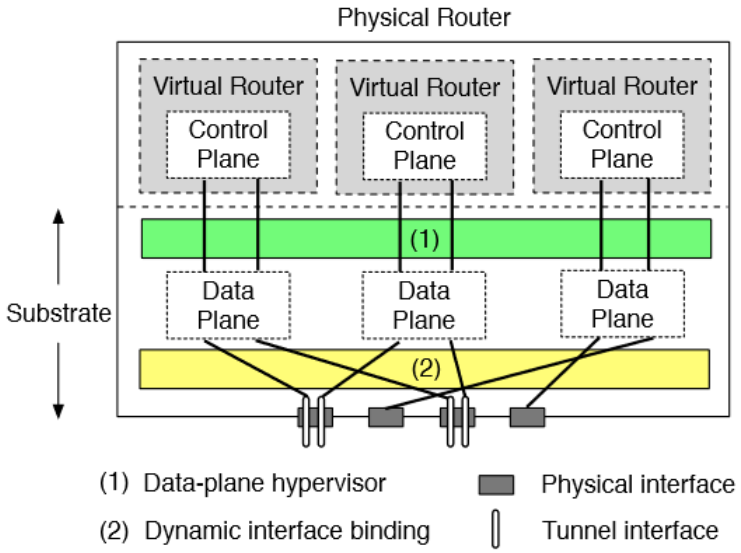

Figure 2.12: Architecture of a VROOM router, from [43].

The migration process itself starts with a tunnel being set between the two routers, in order for the destination router to start receiving routing messages. Afterwards, the control plane will be migrated while traffic is still being handled by the original data plane. With the control plane already running at the new location, the new data plane at the migration destination starts to get populated, while the old data plane can still be updated (these are parallel operations). During this transition, the old router forwards routing protocol traffic to the new router and, once the new data plane is ready, link migration is triggered with both data planes operating simultaneously for a small period, to facilitate the asynchronous link migration. When this task is over the tunnel originally set is closed and the migration is considered finished.

The work by Ghorbani et al. (LIME) [23] explores a similar problem in the context of SDN. Their goal is to migrate a switch and its VMs transparently, i.e., without the controller or the user applications noticing the migration process. Their notion of transparent migration consists in no event happening during a migration that it could not have happened during a normal operation period. This, for example, deems small packet losses as acceptable since they can also occur during normal operation. LIME was designed to fit between the switches and the controller, thus supporting any SDN controller. The authors' approach is similar to that of VROOM, with the use of tunnels between the migration source and target switches, with those working in parallel during the process (fig. 2.13). In addition, the authors include merging mechanisms (e.g., for controller statistics)

to guarantee correctness during the migration.

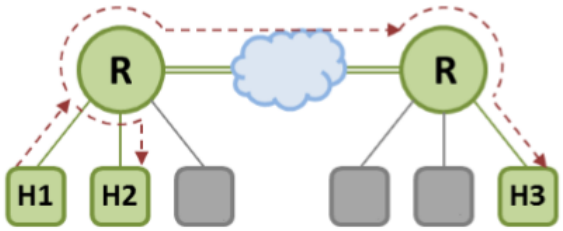

Figure 2.13: Migration source and target switches working in parallel, from [23].

2.5.1 Migration Scheduling

When migrating several elements, potentially with different characteristics, the migration order and timing become key aspects of the migration algorithm, so that it can run swiftly and unnoticed. Therefore, this section is dedicated to some recent works on scheduling of network migration.

We start with [22], where the goal is the sequencing of the migration of various elements that need to be moved when migrating one or more VMs. The starting point is a network consisting of a set of switches (each with its own set of forwarding rules), and the objective is to calculate a sequence of OpenFlow instructions that will virtually move a set of VMs from the starting network into a desired target network, while preserving the desired correctness conditions.

This comes down to solving two sub-problems. The first is defining a sequence of VM migrations (VM sequence planning problem). The second is, for each VM, to determine the order of the OpenFlow instructions that should be applied or discarded (network sequence planning problem).

A way to solve these problems would be to formulate them as a single optimization problem. However, that would result in intolerable computational costs. The solution proposed was a simple heuristic for sequence planning that aimed to fulfil a given bandwidth requirement and avoid cycles. According to their evaluation, the solution is orders of magnitude faster than the optimal algorithm, and achieves approximate results.

The authors of [36] proposed algorithms that aimed to achieve an optimal sequencing for migrations. They proposed three algorithms. First, LMCF (Local Minimum Cost First), an algorithm that attempted to minimize migration cost by migrating one node at a time. Second, MIS-SS (Maximal Independent Set-Size Sequence), whose goal was to minimize migration time by migrating multiple nodes at once. Finally, MIS-LMCF (Maximal Independent Set-Local Minimum Cost First), which tried to minimize migration time and cost.

2.6 Final Considerations

This chapter provided some background and insights for the problem to be addressed next. We presented the earlier machine virtualization and migration techniques and their limitations. We introduced SDNs, and have shown how this new paradigm enabled new network virtualization solutions that fully decouple the virtual network from the substrate. Finally, we have detailed a set of works that started addressing of the migration of certain network elements.

Next, we present our study that aims to compare the trade-offs of different network migration approaches.

Chapter 3

Design and Implementation

The invention of machine virtualization has offered unprecedented flexibility to cloud infrastructures. Virtual machines (VMs) allow a level of mobility and agility that the network infrastructure has been unable to par, and VM migration is now an established technique. However, the ability to move the network together with the VMs that interconnect them is still a mostly unsolved problem.

The emergence of SDNs gave operators new tools to tackle the problem. The global view offered by a logically centralized control plane enabled a new perspective for network virtualization. Modern platforms leverage SDN to fully virtualize a network [32, 12]. These solutions are enablers for network migration, but they do not tackle the problem. In this chapter we start addressing it by presenting three algorithms for network migration. These will then be evaluated and compared in the next chapter.

3.1 Context and Objectives

The context of our work is the SDN-based network virtualization platform Sirius [12]. Sirius runs as an application on top of a SDN controller. It takes all decisions related to the placement of virtual networks, and sets up the network paths by configuring all network switches. This network hypervisor also intercepts messages between the substrate infrastructure and the users' virtual networks, and vice-versa, thus enabling complete network virtualization. This architecture is similar to other virtualization solutions, such as NVP [32], so the algorithms we will evaluate could also be used in those platforms. As with those other solutions, Sirius does not include any module for network migration. This is the gap that our work fills.

Ideally, the migration of a network switch should maintain network performance as close as possible to the one expected during normal operation, from the point of view of applications. Furthermore, the migration process should be transparent to the network control plane, that is, network state should be kept consistent during and after the algorithms' execution. These are the trade-offs we aim to investigate in this work.

3.2 Migration Platform Overview

In Figure 3.1 we present a high level view of our migration platform.

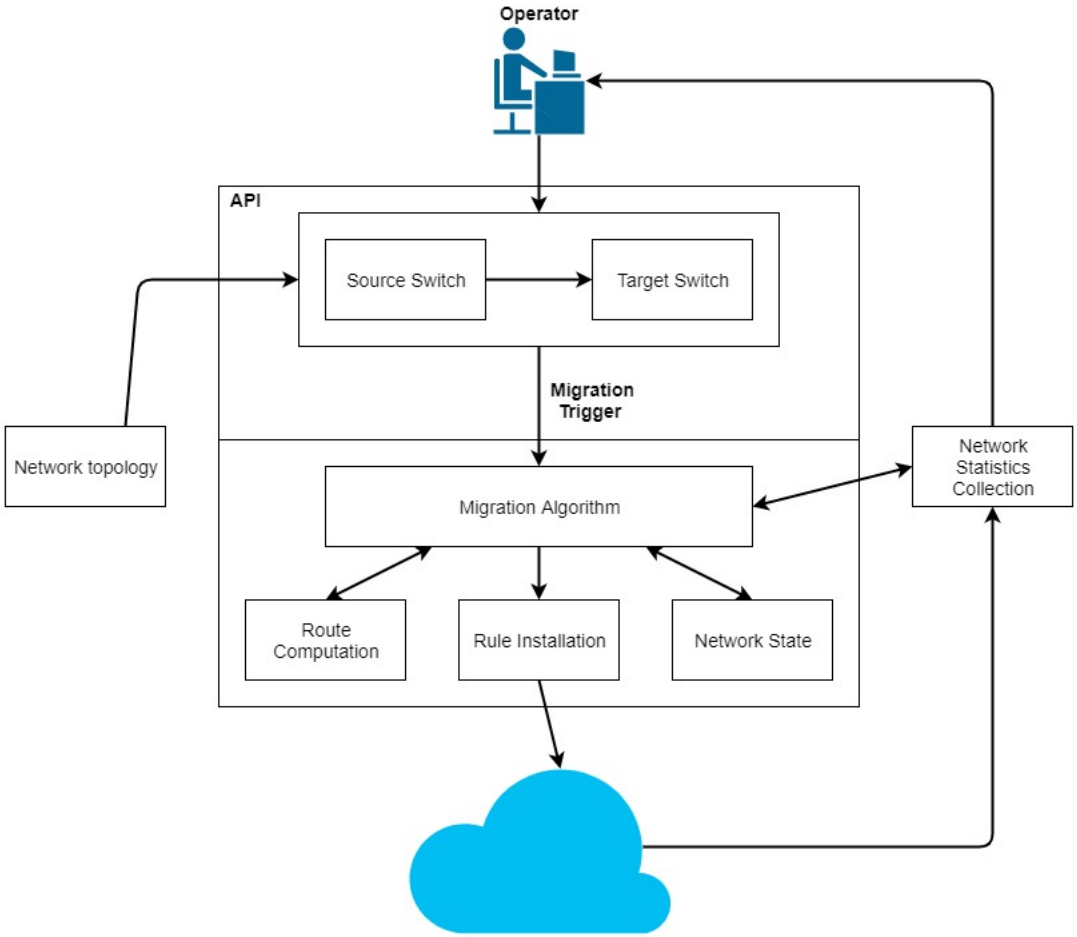

Figure 3.1: High level view of the migration platform.

The user of the platform (e.g., a data center operator) is presented with an API that enables triggering a migration procedure. Its invocation is simple: informed with the network topology, the user identifies the switch that will be migrated (Source Switch), and the migration target, the switch to which its state should be moved to (Target Switch).

The main module of the platform, which forms the core of this thesis, is the Migration Algorithm. We have implemented three different algorithms to analyse different trade-offs. These will be detailed in Section 3.3.

Additionally, we implemented some auxiliary modules, including Route Computation, responsible to return a shortest path given a topology and a pair <source, destination>; Rule Installation, which will translate the outcome of the Migration Algorithm module onto switch rules (made effective with OpenFlow commands); and Network Statistics Collection module, responsible to retrieve up-to-date views of the state

of switches. This module can be used by both the operator, to help decide when to trigger a migration operation, and by the migration algorithms that need to maintain specific switch state during the process. In Section 3.4 we present implementation details of these auxiliary modules.

3.3 Migration Algorithms Under Analysis

This chapter will present the three migration algorithms considered for our platform: *freeze and copy*, *move*, and *clone*. Each of these algorithms makes different assumptions and has different advantages and disadvantages that we hope help illuminate the trade-offs involved in the network migration problem.

To help explain some of the logic behind each algorithm, we will use as example a real topology based on Abilene's core network topology [4], adapted to suit our SDN-based solution, as seen in Figure 3.2. Please note that this topology serves only as an example. In the evaluation we will experiment with a variety of topologies.

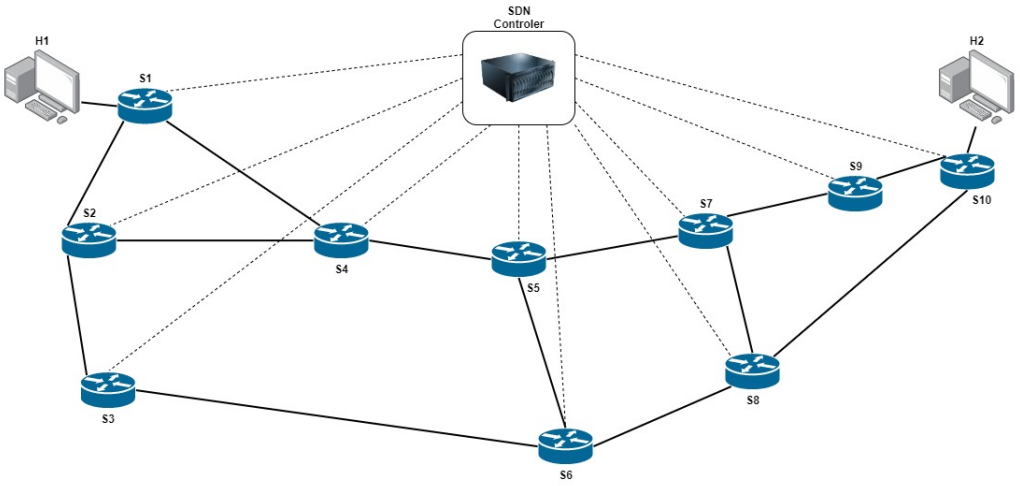

Figure 3.2: Example topology.

3.3.1 Freeze and Copy

The *freeze and copy* algorithm is presented as Algorithm 1. In our example switch S7 will be the Source Switch and switch S6 will be the Target Switch (Fig. 3.3).

The first step (Line 1) is to freeze the source switch. Freezing means the controller stops the source switch from accepting new rules and from deleting old rules. This guarantees that no new flows will be installed in the switch that we want to migrate, so that the switch is effectively *frozen*. The switch can, however, continue to process packets of existing flows.

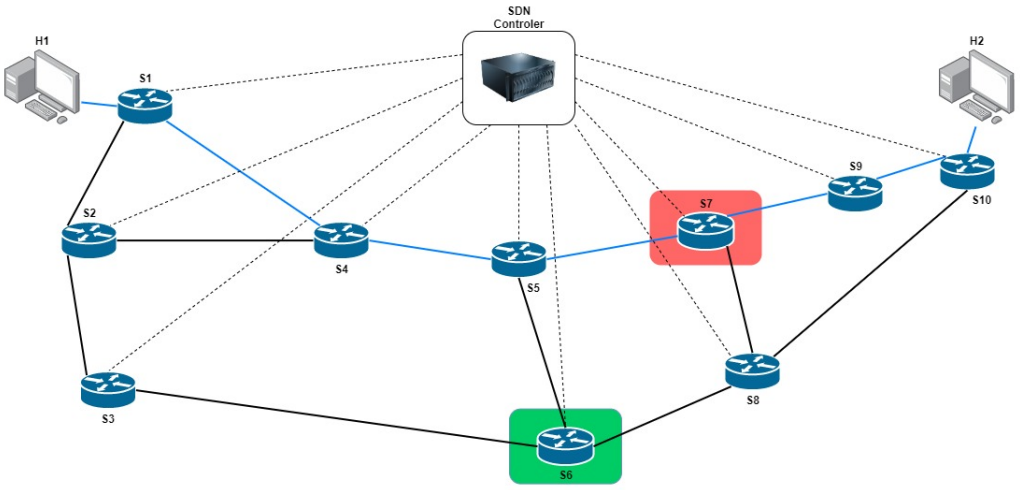

Figure 3.3: Migration switches identified.

Then, in Line 2 we obtain information about the existing flows in the source switch. Each flow can be represented by different identifiers, including <sourceIP, destinationIP>, the 5-tuple <sourceIP, destinationIP, sourcePort, destinationPort, protocol>, or others. In Figure 3.3 you can see highlighted, in blue, an example of a flow between hosts H1 and H2 that would be identified during this step. The network topology is then updated in Line 3 (the source switch is removed, alongside its links).

Algorithm 1: Freeze and copy

input : NetworkTopology, SourceSwitch, TargetSwitch
output: NewNetworkTopology

```
1  Freeze(SourceSwitch);
2  FlowList = FlowTableCollector(SourceSwitch);
3  NewNetworkTopology = deleteSwitch(SourceSwitch, NetworkTopology);
4  foreach flow in FlowList do
5  |   removeFlow(flow, SourceSwitch);
6  |   SP = RouteComputation(flow, NewNetworkTopology);
7  |   foreach switch in SP do
8  |   |   RouteInstallation(switch);
9  |   end
10 end
11 return NewNetworkTopology;
```

With the flows that must be migrated already identified, the controller then removes all existing flows from the source switch (Line 5). After this step the switch starts dropping all packets for the current flow. The new shortest path for each flow is then calculated by using the Dijkstra Algorithm as part of the Route Computation module (Line 6). Finally, the rules of all switches in the new path for the current flow are updated (Line 8). For simplicity, we let some of the old rules to be kept in switches of the previous shortest path,

as they will eventually time out as is typical in production environments. An example result for this process can be seen in Figure 3.4, with the new path highlighted in purple.

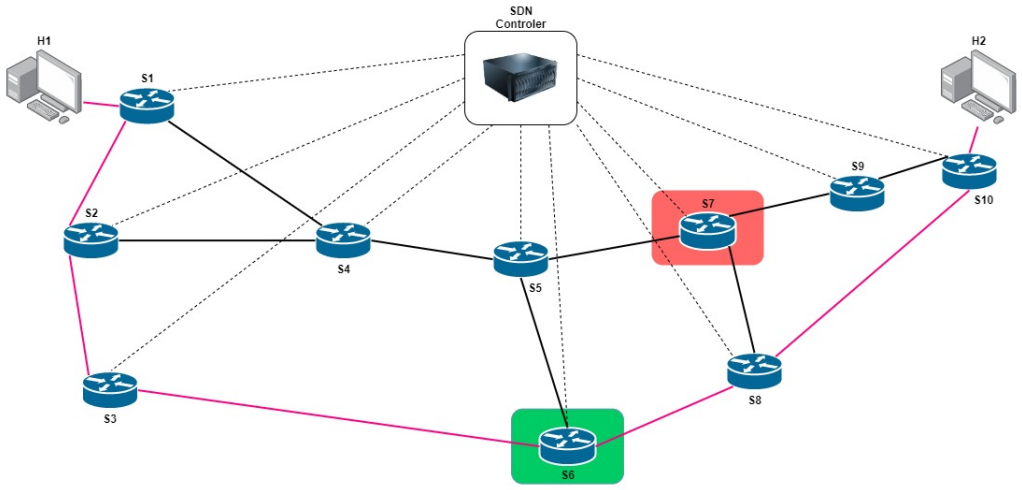

Figure 3.4: Migration result.

The *freeze and copy* algorithm is simple, as it does not require maintaining or processing much state, except the forwarding rules. This simplicity comes at the cost of packet drops during the migration period, and the inability to maintain other state, such as packet counters.

3.3.2 Move

Move is next presented as Algorithm 2. This algorithm is similar to *freeze and copy* with some subtle, but important, differences. In the first step (Line 1) we start again with the freeze procedure, with the controller informing the source switch that it should stop accepting or deleting any rules. Just like previously, the frozen switch keeps processing packets. Then, we retrieve all flows from the forwarding table (Line 2), but, differently from the previous algorithm, we also retrieve other switch state (Line 3), namely traffic counters (this could be further extended with the collection of other stateful elements such as registers). These steps are processed with the help of the Network Statistics Collection module.

Next, in Line 4 we update the topology by removing the Source Switch and its links. Afterwards, the existing flows on Source Switch get deleted (Line 6), and the switch's state is cleared (Line 7). New routes for each flow are calculated in Line 8, with the resulting forwarding rules getting installed (Line 10). Finally, the new switch (Target Switch) gets its state variables updated (Line 11) with the state previously obtained by the Network Statistics Collection.

Algorithm 2: Move

input : NetworkTopology, SourceSwitch, TargetSwitch
output: NewNetworkTopology

```
1  Freeze(SourceSwitch);
2  FlowList = FlowTableCollector(SourceSwitch);
3  NetworkState = NetworkStatisticsCollection(SourceSwitch);
4  NewNetworkTopology = deleteSwitch(SourceSwitch, NetworkTopology);
5  foreach flow in FlowList do
6      removeFlow(flow, SourceSwitch);
7      removeState(flow, SourceSwitch, NetworkState);
8      SP = RouteComputation(flow, NewNetworkTopology);
9      foreach switch in SP do
10         RouteInstallation(flow, switch);
11         updateState(flow, switch, NetworkState);
12     end
13 end
14 return NewNetworkTopology;
```

This algorithm has one particularity that distinguishes it from *freeze and copy*. Namely, it maintains other state besides the flow table. Namely, packet counters, which are in the interest of network operators to maintain important information (e.g. related to accounting, traffic charging, or attacks) that would be lost with the previous algorithm. Overall, we trade-off some complexity for more network state information.

3.3.3 Clone

Finally, we present *clone*, as Algorithm 3. Similarly to the other two, this algorithm starts with the controller freezing the source switch (Line 1). Next, we retrieve both the flows from the source switch's forwarding table (Line 2), and its state (Line 3). In Line 4, we update the topology, just like in *move*.

The main difference is that in *clone* we calculate the new routes for each flow in an earlier stage (Line 6). Afterwards, we update the state of the switches (Line 8), and install the newly calculated routes (Line 9). As we haven't yet removed the rules for the old flows, and have already installed new ones, we have, at this point, two active switches (and paths) for each flow (hence the name clone). An example of this can be seen in Figure 3.5.

Importantly, rule installation in the new path is done in a backwards fashion, from the flow destination to the flow source. So, as the switch closer to the flow source is installed with new rules, the packets for this flow start using the new path, and *no packet is dropped*.

Finally, the old rules are removed from the Source Switch (Line 11), as well as other state (Line 12).

This algorithm has two characteristics that distinguish it from the *move* version. Be-

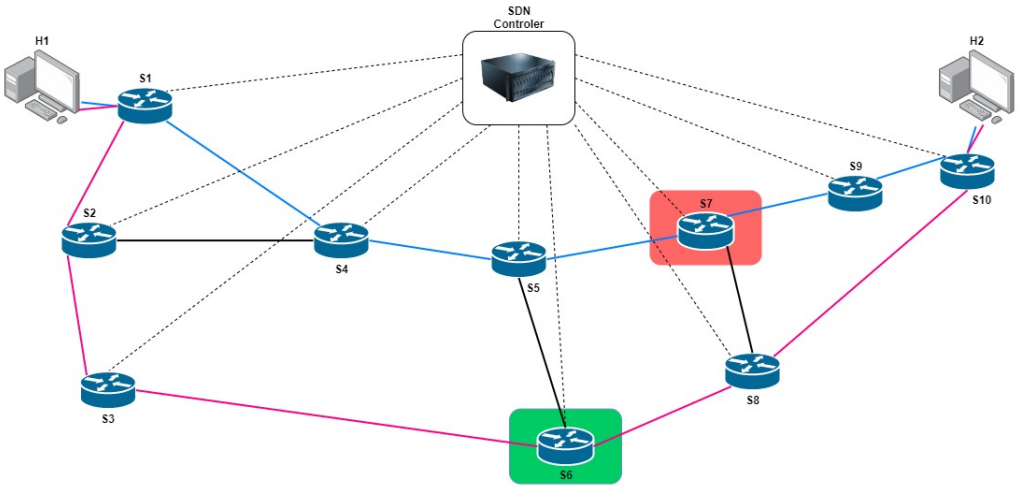

Figure 3.5: Migration with 2 active paths during *clone migration*.

sides maintaining other state besides the flow table, it starts installing the new rules in the new path before removing the old ones from the source switch, in a backward fashion, which helps avoid packet drops, as explained. In addition, as state update is made before route installation, the packet counters are more correct (in the move algorithm some packet counts may be missed in the migration process as the flow is installed before the counters are updated). Overall, we trade off more complexity for better performance and more consistent information, improving transparency.

Algorithm 3: Clone

input : NetworkTopology, SourceSwitch, TargetSwitch
output: NewNetworkTopology

```
1  Freeze(SourceSwitch);
2  FlowList = FlowTableCollector(SourceSwitch);
3  NetworkState = NetworkStatisticsCollection(SourceSwitch);
4  NewNetworkTopology = deleteSwitch(SourceSwitch, NetworkTopology);
5  foreach flow in FlowList do
6      SP = RouteComputation(flow, NewNetworkTopology);
7      foreach switch in SP.backwards() do
8          updateState(flow, switch, NetworkState);
9          RouteInstallation(flow, switch);
10     end
11     removeFlow(flow, SourceSwitch);
12     removeState(flow, SourceSwitch, NetworkState);
13 end
14 return NewNetworkTopology;
```

3.4 Implementation

Our implementation was done on top of the Java-based SDN controller Floodlight [1]. The implementation of the migration algorithms followed the structure of the pseudocode presented in the previous section. In this section we give a bit more detail on the other modules.

3.4.1 Migration Trigger

As mentioned in Section 3.2, the migration algorithms expose an interface to allow operators to trigger a migration process. This interface was implemented as a RESTful API, each linked to the respective methods that implement the logic of the algorithm. The advantage of this type of interface is twofold. First, it is well integrated with Floodlight. Second, it adapts well to different types of remote calls.

3.4.2 Route Computation

Since Floodlight does not provide a module that allows for route computation with proactive rule installation, the SDN approach we followed, we implemented our own module.

The algorithm used by this module to calculate a shortest path implements the Dijkstra's algorithm, returning a list of pairs of nodes (switches). This allows, for example, to easily integrate this algorithm with both the proactive and reactive rule installers that are native to Floodlight.

3.4.3 Rule Installation

Floodlight provides two modules to install forwarding rules in the switches: one reactive and one proactive. As explained, we opted for a proactive approach, so we install the rules generated by the migration algorithms onto the switches using the *Static Entry Pusher*, the native Floodlight module that enables rule installation for proactive settings.

3.4.4 Network Statistics Collection

A Floodlight application can trigger the request of switch statistics, including flow information and packet/byte counters, both periodically and upon request. In our platform we consider only the second option: an operator can request statistics to decide when it should trigger a migration, and the migration algorithm can also request these statistics, as needed. We have only implemented this second option. The Network Statistics Collection module is called to retrieve flow information (such as packet counters) for the *move* and *clone* algorithms, when these data are needed.

This module is able to gather this information in one additional way: by intercepting the Openflow messages generated once a rule gets removed from a switch. This is especially helpful after a migration, since these messages provide the last values before the flows were interrupted. This information is maintained in a data structure we call Network State (see Figure 3.1).

3.5 Summary

This chapter presented the design and implementation details of our network migration platform, with a focus on the migration algorithms and the enabler modules we developed. We considered three algorithms: *freeze and copy*, *move*, and *clone*. This last algorithm was included as a module in the Sirius multi-cloud network hypervisor [12].

Chapter 4

Evaluation

The operation of data centers aims to fulfill specific quality of service metrics to offer a good experience to its users. The evaluation we present in this chapter originates from the consideration of several of these metrics and how the migration process can impact them. Specifically, we investigate how migrating a switch affects application latency, throughput, and packet loss.

4.1 Test Environment

To evaluate our solution, we used a network emulator called mininet [3] (which can emulate an entire network with diverse topologies, including switches and hosts). The emulated network is controlled by the Floodlight controller running our network migration solution. All our scenarios used 2 hosts, generating multiple application flows, each of them connected to a different switch. The random topologies used for evaluation were generated by Boston University's BRITE simulator [39], with the number of nodes varying between 25, 50 and 100. Each node has an average of 3 links with other random nodes.

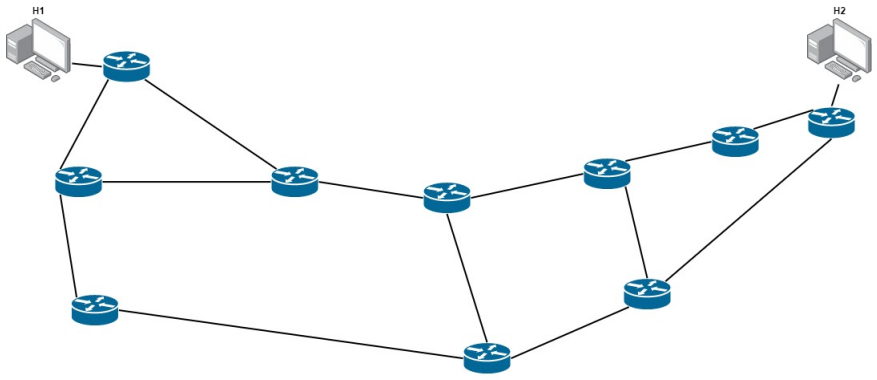

Figure 4.1: Example of a test topology.

All the tests were run in a local environment, with the controller, hosts and network running on the same physical machine (quad-core CPU @3.40GHz with 8Gb DDR3 RAM).

4.2 Duration of Migration Procedure

We start by evaluating the time it takes for each of the three solutions tested to execute the migration procedure. For this purpose we measure the time since the migration is triggered, up until it is finished. We run each experiment 100 times, and present the average and standard deviation. The duration of the execution is presented in Figure 4.2 (lower is better).

As can be observed, for each algorithm, by varying the number of nodes of the topology, the duration increases with topology size, since the module has to recalculate the route to replace the existing one, and more nodes inevitably result in more available paths, which translates in a longer time to run the Dijkstra algorithm. Anyway, the increase is sub-linear to all algorithms, which demonstrates good scalability.

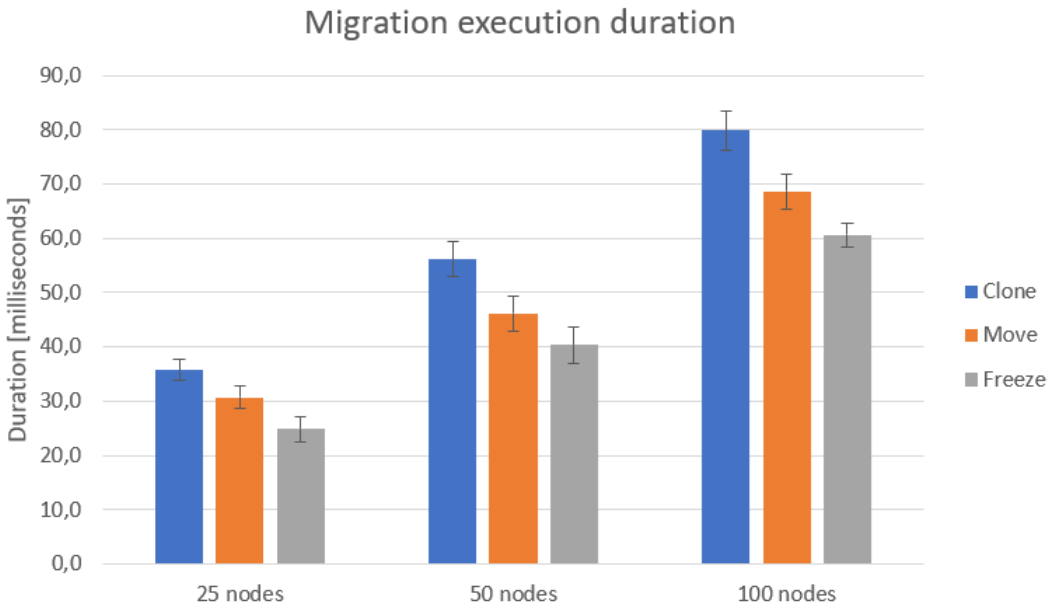

Figure 4.2: Migration time per topology size.

Figure 4.2 also shows that the complexity added to an algorithm increases its processing time, as expected. The *freeze and copy* algorithm has the shortest execution time for every topology size, whilst the longest is always *clone*.

4.3 Data Plane Latency

One of the requirements of data center operation is to provide low latency to its applications. Increases in network latency may deteriorate applications' performance and its proper execution. For that reason, the migration process must not significantly add to network latency.

The data plane latency was obtained using the "ping" tool between the two hosts present in the topology. We ran pings 100 times during normal operation and during a migration procedure. The difference between those two values is the migration induced latency. As can be seen in Figure 4.3 (lower is better), the latency induced by the migration procedure is very small in all cases, in the order of the hundreds of microseconds. Given that the RTTs were in the order of the tens of milliseconds, these can be considered negligible.

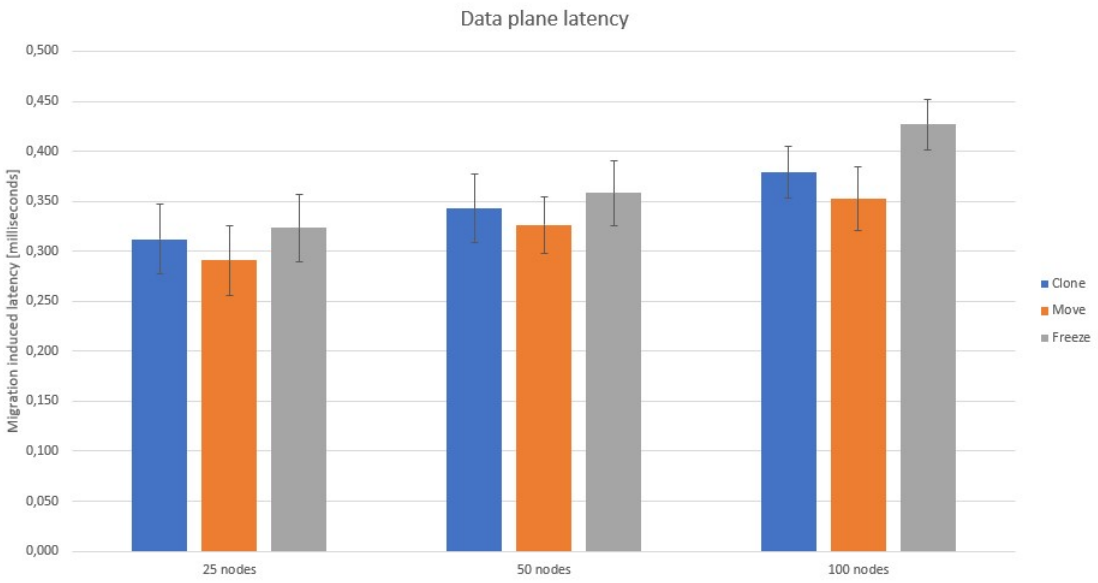

Figure 4.3: Migration induced latency per topology size.

The latency increases, as expected, with the growth of the topology, but not significantly. While the results are all very similar, *freeze and copy* is always slightly worse, and *move* is slightly better. This slightly unexpected result is probably due to minor optimizations in the Java implementation for *move* and *clone*. As the differences were negligible and the delay induced was small anyway, we did not consider adding these optimizations to the *freeze and copy* algorithm.

4.4 Packet Loss

Packet loss is another important metric to assess network operation. In this section, we evaluate the effects of each algorithm on packet loss.

In order to measure the packet loss caused by each of the migration types, we used the "iperf" tool (using the UDP protocol) between the two hosts, set up to use the full available bandwidth of the topology links, while making sure there was no packet loss in normal operation, to enable the analysis of this metric.

We evaluated packet loss by changing both topology size and the number of flow rules to be migrated. The results are shown in Figure 4.4.

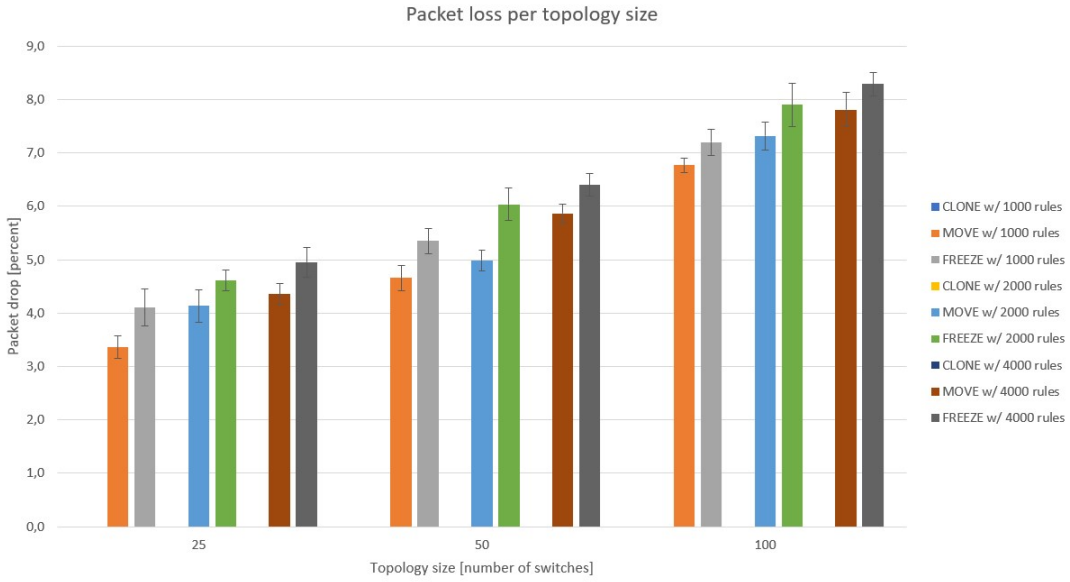

Figure 4.4: Packet loss with varying topology sizes.

The main takeaway from these results is that the *clone* algorithm does not result in *any* packet loss - for that reason the bars do not appear in the graph. This demonstrates the effectiveness of *clone*, as its design choice of maintaining two switches in operation simultaneously, with at least one working path at all times available, allows all packets of a flow to be transmitted without loss.

It is also possible to see that both the size of the target switch's table and the size of the topology influence the packet loss on the other two variants (*move* and *freeze and copy*). The increase in size of the topology results in a greater impact than the number of rules as it implicates more computations and more interactions with network switches.

4.5 Data Plane Throughput

The final metric we consider for evaluation is data plane throughput. The set up for this experiment is the same as for the previous section, and the results are presented in Figure 4.5. In this experiment the migration procedure was triggered at t=5s and it was finalized less than 100 milliseconds later (as per Figure 4.2). The result is presented as 1-second samples.

In this graph it is clear that the *clone* algorithm does not affect throughput. On the other hand, the other algorithms disrupt network traffic during the migration procedure. The *move* algorithm is slightly better than *freeze and copy* because the migration time is smaller, and as a result it induces less packet loss and the throughput is less affected.

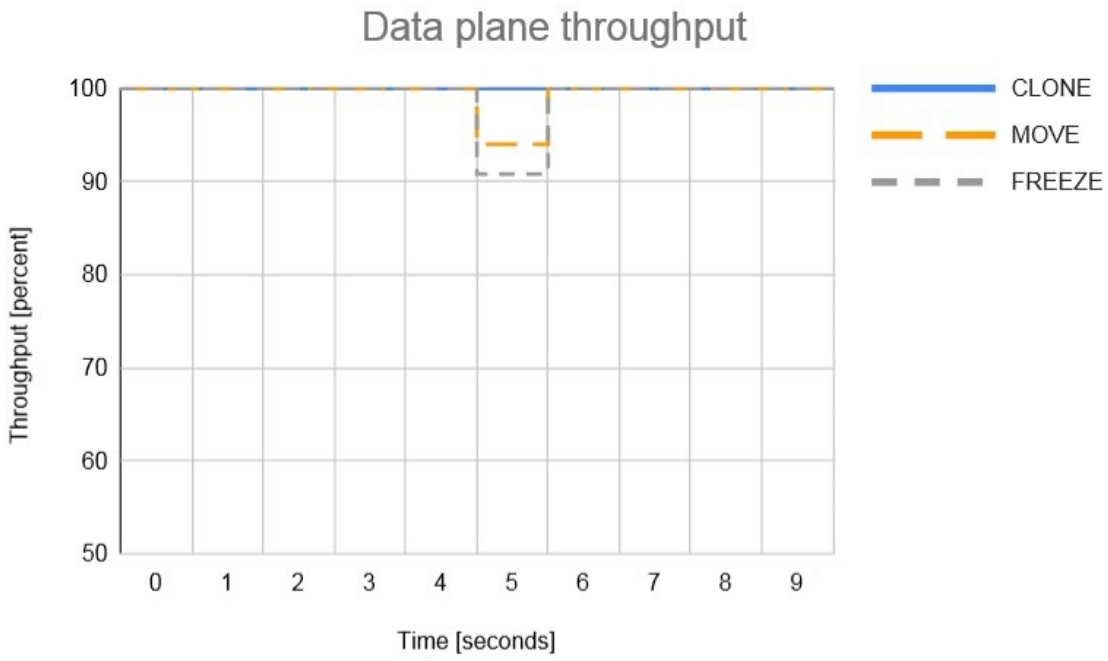

Figure 4.5: Example of a packet loss measurement.

4.6 Summary

This chapter presented the evaluation of the network migration solutions we developed considering common network metrics.

The main takeaway is that the clone algorithm is the one that ensures a higher level of transparency to the network applications. Its additional complexity results in higher migration times, but its design does not induce packet loss, maintaining performance unaffected, the main goal. This is the main justification for its choice as migration algorithm in the Sirius hypervisor.

Chapter 5

Conclusions and Future Work

The way computational resources are managed has evolved in the last decade. Applications moved from running in big bare metal physical servers to the cloud, to improve their flexibility and scalability. This was made possible due to the emergence of virtualization technologies.

In this thesis we studied a mostly unexplored problem: how to migrate a network. Or, more specifically, how to move the state from a network switch (its flow tables and counters) to another switch, without disrupting application performance.

For this purpose we studied, implemented, and evaluated three migration algorithms - *freeze and copy*, *move*, and *clone* - and integrated them into a migration platform based on the Floodlight SDN controller. We discussed the trade-offs of the design of each algorithm, and concluded that an algorithm that is able to maintain a switch "clone" is capable of enabling migration that is transparent to applications. This particular algorithm was included into the Sirius network hypervisor [12].

As future work it would be interesting to integrate this solution with VM migration technologies, to have the entire ensemble of VMs and switches migrated. It would also be interesting to explore the possibility of migrating more complex state (e.g., registers) in the context of programmable data planes.

Bibliography

[1] Floodlight controller. http://www.projectfloodlight.org/floodlight/. Last checked: 17/09/2018.

[2] Google andromeda. https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html. Last checked: 09/10/2019.

[3] Mininet emulator. http://mininet.org/. Last checked: 20/09/2018.

[4] Reference networks. http://www.av.it.pt/anp/on/refnet2.html. Last checked: 27/01/2020.

[5] Vmware nsx. https://www.vmware.com/products/nsx.html. Last checked: 09/10/2019.

[6] Thomas S Afferton, Robert D Doverspike, Charles R Kalmanek, and Kadangode K Ramakrishnan. Packet-aware transport for metro networks. *IEEE Communications Magazine*, 42(3):120–127, 2004.

[7] Mukesh Agrawal, Susan R Bailey, Albert Greenberg, Jorge Pastor, Panagiotis Sebos, Srinivasan Seshan, Kobus Van Der Merwe, and Jennifer Yates. Routerfarm: Towards a dynamic, manageable network edge. In *Proceedings of the 2006 SIGCOMM workshop on Internet network management*, pages 5–10. ACM, 2006.

[8] Geetha Sowjanya Akula and Anupama Potluri. Heuristics for migration with consolidation of ensembles of virtual machines. In *Communication Systems and Networks (COMSNETS), 2014 Sixth International Conference on*, pages 1–4. IEEE, 2014.

[9] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. Openvirtex: Make your virtual sdns programmable. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 25–30. ACM, 2014.

[10] Max Alaluna, Fernando MV Ramos, and Nuno Neves. (literally) above the clouds: Virtualizing the network over multiple clouds. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*, pages 112–115. IEEE, 2016.

[11] Max Alaluna, Eric Vial, Nuno Neves, and Fernando Ramos. Secure and dependable multi-cloud network virtualization. In *Proceedings of the 1st International Workshop on Security and Dependability of Multi-Domain Infrastructures*, page 2. ACM, 2017.

[12] Max Alaluna, Eric Vial, Nuno Neves, and Fernando MV Ramos. Secure multi-cloud network virtualization. *Computer Networks*, 2019.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[14] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[15] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 431–442. ACM, 2012.

[16] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179. ACM, 2007.

[17] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[18] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[19] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[20] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. Live gang migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 135–146. ACM, 2011.

[21] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 51–66, 2018.

[22] Soudeh Ghorbani and Matthew Caesar. Walk the line: consistent network updates with bandwidth guarantees. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 67–72. ACM, 2012.

[23] Soudeh Ghorbani, Cole Schlesinger, Matthew Monaco, Eric Keller, Matthew Caesar, Jennifer Rexford, and David Walker. Transparent, live migration of a software-defined network. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[24] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24. ACM, 2012.

[25] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26. ACM, 2013.

[26] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[27] Qin Jia, Zhiming Shen, Weijia Song, Robbert Van Renesse, and Hakim Weatherspoon. Supercloud: Opportunities and challenges. *ACM SIGOPS Operating Systems Review*, 49(1):137–141, 2015.

[28] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. Smart spot instances for the supercloud. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, page 5. ACM, 2016.

[29] Ian Kash, Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. Economics of a supercloud. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, page 7. ACM, 2016.

[30] Eric Keller, Soudeh Ghorbani, Matt Caesar, and Jennifer Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 109–114. ACM, 2012.

[31] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[32] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan J Jackson, et al. Network virtualization in multi-tenant datacenters. In *NSDI*, volume 14, pages 203–216, 2014.

[33] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[34] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[35] Tao Lei, Zhaoming Lu, Xiangming Wen, Xing Zhao, and Luhan Wang. Swan: An sdn based campus wlan framework. In *Wireless Communications, Vehicular Technology, Information Theory and Aerospace & Electronic Systems (VITAE), 2014 4th International Conference on*, pages 1–5. IEEE, 2014.

[36] Samantha Lo, Mostafa Ammar, and Ellen Zegura. Design and analysis of schedules for virtual network migration. In *IFIP Networking Conference, 2013*, pages 1–9. IEEE, 2013.

[37] Diogo Menezes Ferrazani Mattos and Otto Carlos Muniz Bandeira Duarte. Xenflow: Seamless migration primitive and quality of service for virtual networks. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 2326–2331. IEEE, 2014.

[38] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[39] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. Brite: Universal topology generation from a user's perspective. Technical report, Boston University Computer Science Department, 2001.

[40] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 1:132, 2009.

[41] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[42] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[43] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus Van Der Merwe, and Jennifer Rexford. Virtual routers on the move: live router migration as a network-management primitive. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 231–242. ACM, 2008.

[44] Dan Williams, Hani Jamjoom, Zhefu Jiang, and Hakim Weatherspoon. Virtualwires for live migrating virtual networks across clouds. *Report by Cornell University and IBM TJ Watson Research Center, New York*, 2013.

[45] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The xen-blanket: virtualize once, run everywhere. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 113–126. ACM, 2012.

[46] Timothy Wood, KK Ramakrishnan, Prashant Shenoy, Jacobus Van Der Merwe, Jinho Hwang, Guyue Liu, and Lucas Chaufournier. Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines. *IEEE/ACM Transactions on Networking (TON)*, 23(5):1568–1583, 2015.

[47] Timothy Wood, Prashant J Shenoy, Alexandre Gerber, Jacobus E van der Merwe, and Kadangode K Ramakrishnan. The case for enterprise-ready virtual private clouds. In *HotCloud*, 2009.

[48] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445. ACM, 2017.

[49] Kejiang Ye, Xiaohong Jiang, Ran Ma, and Fengxi Yan. Vc-migration: Live migration of virtual clusters in the cloud. In *Proceedings of the 2012 ACM/IEEE 13th*

International Conference on Grid Computing, pages 209–218. IEEE Computer Society, 2012.

[50] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 41(4):351–362, 2011.